

Our Ref.: 723-963

U.S. PATENT APPLICATION

Inventors: Farhad FOULADI
Mark M. LEATHER
Robert MOORE
Howard CHENG
Timothy J. VAN HOOK

Invention: GRAPHICS SYSTEM WITH COPY OUT CONVERSIONS BETWEEN
EMBEDDED FRAME BUFFER AND MAIN MEMORY

***NIXON & VANDERHYE P.C.
ATTORNEYS AT LAW
1100 NORTH GLEBE ROAD
8TH FLOOR
ARLINGTON, VIRGINIA 22201-4714
(703) 816-4000
Facsimile (703) 816-4100***

SPECIFICATION

Specification

GRAPHICS SYSTEM WITH COPY OUT CONVERSIONS BETWEEN EMBEDDED FRAME BUFFER AND MAIN MEMORY

5

Cross-Reference to Related Applications

This application is filed in accordance with 35 U.S.C. §119(e)(1) and claims the benefit of the provisional application Serial No. 60/227,030 filed on August 23, 2000, entitled "Graphics System With Copy Out Conversions Between
10 Embedded Frame Buffer And Main Memory."

5ws
A1
This application is related to the following co-pending applications identified below (by title and attorney docket number), which focus on various aspects of the graphics system described herein. Each of the following applications are hereby incorporated herein by reference.

- 15 • provisional Application No. 60/161,915, filed October 28, 1999 and its corresponding utility Application No. 09/465,754, filed December 17, 1999, both entitled "Vertex Cache For 3D Computer Graphics",
- provisional Application No. 60/226,912, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 20 723-959), both entitled "Method and Apparatus for Buffering Graphics Data in a Graphics System ",

- provisional Application No. 60/226,889, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-958), both entitled "Graphics Pipeline Token Synchronization",
- provisional Application No. 60/226,891, filed August 23, 2000 and its
5 corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-961), both entitled "Method And Apparatus For Direct and Indirect Texture Processing In A Graphics System",
- provisional Application No. 60/226,888, filed August 23, 2000 and its
10 corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-968), both entitled "Recirculating Shade Tree Blender For A Graphics System",
- provisional Application No. 60/226,892, filed August 23, 2000 and its
15 corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-960), both entitled "Method And Apparatus For Efficient Generation Of Texture Coordinate Displacements For Implementing Emboss-Style Bump Mapping In A Graphics Rendering System",
- provisional Application No. 60/226,893, filed August 23, 2000 and its
20 corresponding utility Application No. _____ filed _____ (atty. dkt. no. 723-962), both entitled "Method And Apparatus For Environment-Mapped Bump-Mapping In A Graphics System",

- provisional Application No. 60/227,007, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-967), both entitled "Achromatic Lighting in a Graphics System and Method",
- 5 • provisional Application No. 60/226,900, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-964), both entitled "Method And Apparatus For Anti-Aliasing In A Graphics System",
- 10 • provisional Application No. 60/226,910, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-957), both entitled "Graphics System With Embedded Frame Buffer Having Reconfigurable Pixel Formats",
- utility Application No. 09/585,329, filed June 2, 2000, entitled "Variable Bit Field Color Encoding" (atty. dkt. no. 723-749),
- 15 • provisional Application No. 60/226,890, filed August 23, 2000 and its corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-956), both entitled "Method And Apparatus For Dynamically Reconfiguring The Order Of Hidden Surface Processing Based On Rendering Mode",
- 20 • provisional Application No. 60/226,915, filed August 23, 2000 and its

corresponding utility Application No. _____ filed _____ (atty. dkt. no. 723-973), both entitled "Method And Apparatus For Providing Non-Photorealistic Cartoon Outlining Within A Graphics System",

- provisional Application No. 60/227,032, filed August 23, 2000 and its
5 corresponding utility Application No. _____, filed _____, (atty. dkt. no. 723-954), both entitled "Method And Apparatus For Providing Improved Fog Effects In A Graphics System",

- provisional Application No. 60/226,885, filed August 23, 2000 and its
10 corresponding utility Application No. _____, filed _____, (atty. dkt. no. 723-969), both entitled "Controller Interface For A Graphics System",

- provisional Application No. 60/227,033, filed August 23, 2000 and its
corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-955), both entitled "Method And Apparatus For Texture Tiling In A Graphics System",

- 15 • provisional Application No. 60/226,899, filed August 23, 2000 and its
corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-971), both entitled "Method And Apparatus For Pre-Caching Data In Audio Memory",

- provisional Application No. 60/226,913, filed August 23, 2000 and its
20 corresponding utility Application No. _____, filed _____ (atty. dkt. no.

723-965), both entitled "Z-Texturing",

- provisional Application No. 60/227,031, filed August 23, 2000 entitled "Application Program Interface for a Graphics System" (atty. dkt. no. 723-880),
- provisional Application No. 60/226,886, filed August 23, 2000 and its
5 corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-970), both entitled "Method and Apparatus for Accessing Shared Resources",
- provisional Application No. 60/226,884, filed August 23, 2000 and its
10 corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-972), both entitled "External Interfaces For A 3D Graphics and Audio Coprocessor",
- provisional Application No. 60/226,894, filed August 23, 2000 and its
15 corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-974), both entitled "Graphics Processing System With Enhanced Memory Controller",
- provisional Application No. 60/226,914, filed August 23, 2000 and its
20 corresponding utility Application No. _____, filed _____, (atty. dkt. no. 723-966), both entitled " Low Cost Graphics System With Stitching Hardware Support For Skeletal Animation", and
- provisional Application No. 60/227,006, filed August 23, 2000 and its

corresponding utility Application No. _____, filed _____ (atty. dkt. no. 723-953), both entitled " Shadow Mapping In A Low Cost Graphics System".

Field of the Invention

The present invention relates to computer graphics, and more particularly to interactive graphics systems such as home video game platforms. Still more particularly this invention relates to a graphics system having a copy pipeline capable of performing various conversion and other processing operations when transferring image data between a reconfigurable embedded frame buffer and main memory. The copy pipeline is used to selectively further process and selectively transfer image data to a display buffer or a texture buffer in main memory.

Background And Summary Of The Invention

Many of us have seen films containing remarkably realistic dinosaurs, aliens, animated toys and other fanciful creatures. Such animations are made possible by computer graphics. Using such techniques, a computer graphics artist can specify how each object should look and how it should change in appearance over time, and a computer then models the objects and displays them on a display such as your television or a computer screen. The computer takes care of performing the many tasks required to make sure that each part of the displayed image is colored and shaped just right based on the position and orientation of each object in a scene, the direction in which light seems to strike each object, the surface texture of each object, and other factors.

Because computer graphics generation is complex, computer-generated three-dimensional graphics just a few years ago were mostly limited to expensive specialized flight simulators, high-end graphics workstations and supercomputers.

The public saw some of the images generated by these computer systems in movies and expensive television advertisements, but most of us couldn't actually interact with the computers doing the graphics generation. All this has changed with the availability of relatively inexpensive 3D graphics platforms such as, for example, the Nintendo 64® and various 3D graphics cards now available for personal computers. It is now possible to interact with exciting 3D animations and simulations on relatively inexpensive computer graphics systems in your home or office.

A problem graphics system designers confronted in the past was to provide a powerful yet inexpensive system which enables various data formats to be stored and processed thereby in a efficient and advantageous manner. Graphics chips used in graphics systems have included a local or on-chip memory for storing data as it is rendered by the graphics pipeline. When data is generated by the graphics chip it is transferred from the local memory to an external memory, where it can be used by, for example, a video interface unit to display the data on a display device. This external memory is typically part of the main memory of the graphics system and is referred to as the external frame buffer (XFB). The processing path of the data between the local memory and the external frame buffer may be referred to as the copy pipeline.

The local memory and the external frame buffer can have a variety of data formats for achieving various functionality in the graphics system. One problem that graphics system designers have faced in the past is to determine what format(s) of data to support in the local memory and the external frame buffer to enable advantageous and efficient use thereof by applications running on the system. Another problem graphics system designers have faced in the past is to find useful and efficient ways in which to maximize the speed, flexibility and

overall operation of system. For example, one problem is to determine what elements in the system should be used to perform certain processes or functions and at what time such processes should be performed. Another problem relates to how to best take advantage of processing pipelines, such as the copy pipeline,
5 which has been typically used to transfer data from a local memory of a graphics chip to the main memory of the system.

Various solutions to these problem were offered. For example, graphics systems have used a variety of data formats and have performed conversions between such formats at various points in the graphics processing operation, in an
10 attempt to improve or maximize the overall operation of the system. While some work has been done in the past in connection with such memories, data formats and conversions further improvements are desirable. Specifically, further improvements are desired for high performance, low cost graphics systems, such as home video game systems.

15 The present invention addresses this problem by providing techniques and arrangements for use in connection with copying out data from an embedded frame buffer to main memory in a graphics system. The invention provides a copy out pipeline which advantageously enables further processing of data and/or data format conversions to be performed "on the fly" during the transfer of the data
20 from the embedded frame buffer to an external destination, such as main memory. The invention further provides a copy pipeline that enables various pixel data formats to be advantageously used in the copy out operation. In addition, the copy pipeline of the invention enables data from the embedded frame buffer to be copied to main memory in a specific format for display or in a variety of texture
25 formats for subsequent use as a texture in a graphics pipeline operation. The copy pipeline of the instant invention is particularly advantageous when used in systems

designed for playing interactive 3D video games. In accordance with the instant invention, the embedded frame buffer can be reconfigured to and efficiently used in a variety of modes, including an anti-aliasing mode, a deflicker mode and a YUV (i.e. luma/chroma) mode, thereby increasing the flexibility of the system to support a variety of applications. The desired pixel format for each mode can be selected using, for example, a command to the graphics hardware on which the embedded frame buffer is provided. The copy pipeline can process and selectively further convert data in any of the formats supported by the embedded frame buffer.

In accordance with the invention, the copy pipeline is advantageously used to further process the data from the embedded frame buffer prior to storing the data in the external frame buffer. For example, the copy pipeline can be used to convert the data between a variety of useful formats to, for example, reduce the amount of memory needed to store the data, and/or provide the data in desired format for use in further processing by the graphics system. The copy pipeline can also be used to further process the frame data in a manner that improves the display quality and/or modifies the display characteristics.

In accordance with one aspect provided by the invention, the graphics system, includes an embedded frame buffer and a copy pipeline which transfers data from the embedded frame buffer to an external location, wherein the copy pipeline converts the data from one format to another format prior to writing the data to the external location. The external location may be a display buffer or a texture buffer in the main memory of the graphics system. The copy pipeline converts the data to a display format if the data is transferred to the display buffer and a texture format if the data is transferred to the texture buffer. The graphics pipeline is operable to use the data in the texture buffer during a subsequent rendering process. The copy pipeline selectively reads data from the embedded

frame buffer in various RGB color formats or a YUV color format, and writes data to main memory in either a display format or a variety of texture formats.

In accordance with another aspect of the invention, a method of transferring data from a graphics chip to an external destination is provided. The method includes storing image data in an embedded frame buffer of the graphics chip, initiating a copy out operation for transferring data from the embedded frame buffer to the external destination, converting the data from one format to another format during the copy out operation, and writing the converted data to the external destination. The method includes either converting the data to a texture format and writing the texture format data to a texture buffer, or converting the data to a display format and writing the display format data to a display buffer. The copy pipeline is also operable to selectively perform scaling, gamma correction and/or anti-aliasing operations during the copy out process and prior to writing the data to main memory. The converting step includes performing at least one of the following conversions: RGB color format to another RGB color format; YUV color format to another YUV color format; RGB color format to YUV color format; and YUV color format to RGB color format.

Brief Description Of The Drawings

These and other features and advantages provided by the invention will be better and more completely understood by referring to the following detailed description of presently preferred embodiments in conjunction with the drawings, of which:

Figure 1 is an overall view of an example interactive computer graphics system;

Figure 2 is a block diagram of the Figure 1 example computer graphics system;

5 Figure 3 is a block diagram of the example graphics and audio processor shown in Figure 2;

Figure 4 is a block diagram of the example 3D graphics processor shown in Figure 3;

10 Figure 5 is an example logical flow diagram of the Figure 4 graphics and audio processor;

Figure 6 shows an exemplary embedded frame buffer and two RGB based formats used herein;

Figure 7 shows the exemplary frame buffer of Figure 6 reconfigured to store YUV formatted data;

15 Figure 8 shows an exemplary interface between the pixel engine and the embedded frame buffer;

Figure 9 shows an exemplary block diagram of the copy pipeline;

Figure 10a illustrates an exemplary conversion of RGB format to YUV format;

20 Figure 10b shows an exemplary block diagram of the data paths used for vertical scaling;

Figure 11 is a block diagram of the graphics system showing how data moves between the various elements of the system;

Figure 12 shows a more detailed block diagram of the copy pipeline of Figure 9;

Figure 12a shows an exemplary 7 tap vertical filter used for anti-aliasing in accordance with a preferred embodiment of the instant invention;

5 Figure 12b shows a block diagram of the anti-aliasing buffering used in accordance with a preferred embodiment of the instant invention;

Figure 12c shows the vertical filter of Figure 12a used for de-flickering;

Figure 12d shows a block diagram of the de-flickering buffering used in accordance with a preferred embodiment of the instant invention;

10 Figure 12e shows a conversion matrix in accordance with a preferred embodiment of the invention;

Figure 12f shows texture tiles being stored in main memory;

Figure 13 illustrates down sampling conversion from YUV444 format to YUV422 format;

15 Figures 14a and 14b illustrate two conversions used to convert data from YUV420 format to YUV444 format;

Figure 15 shows an exemplary control register for use in defining a current pixel format;

20 Figure 16 shows an exemplary register used in connection with a texture copy command; and

Figure 17 shows an exemplary register used in connection with a display copy command; and

Figures 18A and 18B show example alternative compatible implementations.

Detailed Description Of Example Embodiments Of The Invention

Figure 1 shows an example interactive 3D computer graphics system 50.

5 System 50 can be used to play interactive 3D video games with interesting stereo sound. It can also be used for a variety of other applications.

In this example, system 50 is capable of processing, interactively in real time, a digital representation or model of a three-dimensional world. System 50 can display some or all of the world from any arbitrary viewpoint. For example,
 10 system 50 can interactively change the viewpoint in response to real time inputs from handheld controllers 52a, 52b or other input devices. This allows the game player to see the world through the eyes of someone within or outside of the world. System 50 can be used for applications that do not require real time 3D interactive display (e.g., 2D display generation and/or non-interactive display), but
 15 the capability of displaying quality 3D images very quickly can be used to create very realistic and exciting game play or other graphical interactions.

To play a video game or other application using system 50, the user first connects a main unit 54 to his or her color television set 56 or other display device by connecting a cable 58 between the two. Main unit 54 produces both video
 20 signals and audio signals for controlling color television set 56. The video signals are what controls the images displayed on the television screen 59, and the audio signals are played back as sound through television stereo loudspeakers 61L, 61R.

The user also needs to connect main unit 54 to a power source. This power source may be a conventional AC adapter (not shown) that plugs into a standard

home electrical wall socket and converts the house current into a lower DC voltage signal suitable for powering the main unit 54. Batteries could be used in other implementations.

The user may use hand controllers 52a, 52b to control main unit 54.

5 Controls 60 can be used, for example, to specify the direction (up or down, left or right, closer or further away) that a character displayed on television 56 should move within a 3D world. Controls 60 also provide input for other applications (e.g., menu selection, pointer/cursor control, etc.). Controllers 52 can take a variety of forms. In this example, controllers 52 shown each include controls 60
10 such as joysticks, push buttons and/or directional switches. Controllers 52 may be connected to main unit 54 by cables or wirelessly via electromagnetic (e.g., radio or infrared) waves.

To play an application such as a game, the user selects an appropriate storage medium 62 storing the video game or other application he or she wants to
15 play, and inserts that storage medium into a slot 64 in main unit 54. Storage medium 62 may, for example, be a specially encoded and/or encrypted optical and/or magnetic disk. The user may operate a power switch 66 to turn on main unit 54 and cause the main unit to begin running the video game or other application based on the software stored in the storage medium 62. The user may
20 operate controllers 52 to provide inputs to main unit 54. For example, operating a control 60 may cause the game or other application to start. Moving other controls 60 can cause animated characters to move in different directions or change the user's point of view in a 3D world. Depending upon the particular software stored within the storage medium 62, the various controls 60 on the controller 52 can
25 perform different functions at different times.

Example Electronics of Overall System

Figure 2 shows a block diagram of example components of system 50. The primary components include:

- a main processor (CPU) 110,
- a main memory 112, and
- a graphics and audio processor 114.

In this example, main processor 110 (e.g., an enhanced IBM Power PC 750) receives inputs from handheld controllers 108 (and/or other input devices) via graphics and audio processor 114. Main processor 110 interactively responds to user inputs, and executes a video game or other program supplied, for example, by external storage media 62 via a mass storage access device 106 such as an optical disk drive. As one example, in the context of video game play, main processor 110 can perform collision detection and animation processing in addition to a variety of interactive and control functions.

In this example, main processor 110 generates 3D graphics and audio commands and sends them to graphics and audio processor 114. The graphics and audio processor 114 processes these commands to generate interesting visual images on display 59 and interesting stereo sound on stereo loudspeakers 61R, 61L or other suitable sound-generating devices.

Example system 50 includes a video encoder 120 that receives image signals from graphics and audio processor 114 and converts the image signals into analog and/or digital video signals suitable for display on a standard display device such as a computer monitor or home color television set 56. System 50 also includes an audio codec (compressor/decompressor) 122 that compresses and decompresses digitized audio signals and may also convert between digital and

analog audio signaling formats as needed. Audio codec 122 can receive audio inputs via a buffer 124 and provide them to graphics and audio processor 114 for processing (e.g., mixing with other audio signals the processor generates and/or receives via a streaming audio output of mass storage access device 106).

- 5 Graphics and audio processor 114 in this example can store audio related information in an audio memory 126 that is available for audio tasks. Graphics and audio processor 114 provides the resulting audio output signals to audio codec 122 for decompression and conversion to analog signals (e.g., via buffer amplifiers 128L, 128R) so they can be reproduced by loudspeakers 61L, 61R.

- 10 Graphics and audio processor 114 has the ability to communicate with various additional devices that may be present within system 50. For example, a parallel digital bus 130 may be used to communicate with mass storage access device 106 and/or other components. A serial peripheral bus 132 may communicate with a variety of peripheral or other devices including, for example:

- 15
- a programmable read-only memory and/or real time clock 134,
 - a modem 136 or other networking interface (which may in turn connect system 50 to a telecommunications network 138 such as the Internet or other digital network from/to which program instructions and/or data can be downloaded or uploaded), and
- 20
- flash memory 140.

A further external serial bus 142 may be used to communicate with additional expansion memory 144 (e.g., a memory card) or other devices. Connectors may be used to connect various devices to busses 130, 132, 142.

Example Graphics And Audio Processor

Figure 3 is a block diagram of an example graphics and audio processor

114. Graphics and audio processor 114 in one example may be a single-chip ASIC (application specific integrated circuit). In this example, graphics and audio processor 114 includes:

- a processor interface 150,
- a memory interface/controller 152,
- a 3D graphics processor 154,
- an audio digital signal processor (DSP) 156,
- an audio memory interface 158,
- an audio interface and mixer 160,
- a peripheral controller 162, and
- a display controller 164.

3D graphics processor 154 performs graphics processing tasks. Audio digital signal processor 156 performs audio processing tasks. Display controller 164 accesses image information from main memory 112 and provides it to video encoder 120 for display on display device 56. Audio interface and mixer 160 interfaces with audio codec 122, and can also mix audio from different sources (e.g., streaming audio from mass storage access device 106, the output of audio DSP 156, and external audio input received via audio codec 122). Processor interface 150 provides a data and control interface between main processor 110 and graphics and audio processor 114.

Memory interface 152 provides a data and control interface between graphics and audio processor 114 and memory 112. In this example, main processor 110 accesses main memory 112 via processor interface 150 and memory

interface 152 that are part of graphics and audio processor 114. Peripheral controller 162 provides a data and control interface between graphics and audio processor 114 and the various peripherals mentioned above. Audio memory interface 158 provides an interface with audio memory 126.

5 **Example Graphics Pipeline**

Figure 4 shows a more detailed view of an example 3D graphics processor 154. 3D graphics processor 154 includes, among other things, a command processor 200 and a 3D graphics pipeline 180. Main processor 110 communicates streams of data (e.g., graphics command streams and display lists) to command processor 200. Main processor 110 has a two-level cache 115 to minimize memory latency, and also has a write-gathering buffer 111 for uncached data streams targeted for the graphics and audio processor 114. The write-gathering buffer 111 collects partial cache lines into full cache lines and sends the data out to the graphics and audio processor 114 one cache line at a time for maximum bus usage.

Command processor 200 receives display commands from main processor 110 and parses them -- obtaining any additional data necessary to process them from shared memory 112. The command processor 200 provides a stream of vertex commands to graphics pipeline 180 for 2D and/or 3D processing and rendering. Graphics pipeline 180 generates images based on these commands. The resulting image information may be transferred to main memory 112 for access by display controller/video interface unit 164 -- which displays the frame buffer output of pipeline 180 on display 56.

Figure 5 is a logical flow diagram of graphics processor 154. Main processor 110 may store graphics command streams 210, display lists 212 and

vertex arrays 214 in main memory 112, and pass pointers to command processor 200 via bus interface 150. The main processor 110 stores graphics commands in one or more graphics first-in-first-out (FIFO) buffers 210 it allocates in main memory 110. The command processor 200 fetches:

- command streams from main memory 112 via an on-chip FIFO memory buffer 216 that receives and buffers the graphics commands for synchronization/flow control and load balancing,
- display lists 212 from main memory 112 via an on-chip call FIFO memory buffer 218, and
- vertex attributes from the command stream and/or from vertex arrays 214 in main memory 112 via a vertex cache 220.

Command processor 200 performs command processing operations 200a that convert attribute types to floating point format, and pass the resulting complete vertex polygon data to graphics pipeline 180 for rendering/rasterization. A programmable memory arbitration circuitry 130 (see Figure 4) arbitrates access to shared main memory 112 between graphics pipeline 180, command processor 200 and display controller/video interface unit 164.

Figure 4 shows that graphics pipeline 180 may include:

- a transform unit 300,
- a setup/rasterizer 400,
- a texture unit 500,
- a texture environment unit 600, and
- a pixel engine 700.

Transform unit 300 performs a variety of 2D and 3D transform and other operations 300a (see Figure 5). Transform unit 300 may include one or more matrix memories 300b for storing matrices used in transformation processing 300a. Transform unit 300 transforms incoming geometry per vertex from object space to screen space; and transforms incoming texture coordinates and computes projective texture coordinates (300c). Transform unit 300 may also perform polygon clipping/culling 300d. Lighting processing 300e also performed by transform unit 300b provides per vertex lighting computations for up to eight independent lights in one example embodiment. Transform unit 300 can also perform texture coordinate generation (300c) for embossed type bump mapping effects, as well as polygon clipping/culling operations (300d).

Setup/rasterizer 400 includes a setup unit which receives vertex data from transform unit 300 and sends triangle setup information to one or more rasterizer units (400b) performing edge rasterization, texture coordinate rasterization and color rasterization.

Texture unit 500 (which may include an on-chip texture memory (TMEM) 502) performs various tasks related to texturing including for example:

- retrieving textures 504 from main memory 112,
- texture processing (500a) including, for example, multi-texture handling, post-cache texture decompression, texture filtering, embossing, shadows and lighting through the use of projective textures, and BLIT with alpha transparency and depth,
- bump map processing for computing texture coordinate displacements for bump mapping, pseudo texture and texture tiling effects (500b), and

- indirect texture processing (500c).

Texture unit 500 outputs filtered texture values to the texture environment unit 600 for texture environment processing (600a). Texture environment unit 600 blends polygon and texture color/alpha/depth, and can also perform texture fog processing (600b) to achieve inverse range based fog effects. Texture environment unit 600 can provide multiple stages to perform a variety of other interesting environment-related functions based for example on color/alpha modulation, embossing, detail texturing, texture swapping, clamping, and depth blending..

Pixel engine 700 performs depth (z) compare (700a) and pixel blending (700b). In this example, pixel engine 700 stores data into an embedded (on-chip) frame buffer memory 702. Graphics pipeline 180 may include one or more embedded DRAM memories 702 to store frame buffer and/or texture information locally. Z compares 700a' can also be performed at an earlier stage in the graphics pipeline 180 depending on the rendering mode currently in effect (e.g., z compares can be performed earlier if alpha testing is not required). The pixel engine 700 includes a copy operation 700c that periodically writes on-chip frame buffer 702 to main memory 112 for access by display/video interface unit 164. This copy operation 700c can also be used to copy embedded frame buffer 702 contents to textures in the main memory 112 for dynamic texture synthesis effects. Anti-aliasing and other filtering can be performed during the copy-out operation. The frame buffer output of graphics pipeline 180 (which is ultimately stored in main memory 112) is read each frame by display/video interface unit 164. Display controller/video interface 164 provides digital RGB pixel values for display on display 102.

Example Configurations for the Embedded Frame Buffer

As generally shown in Fig. 4, the embedded frame buffer 702 receives data from the graphics pipeline 180. The graphics pipeline renders primitives in

5 RGB(A) format. Thus, as will be explained in more detail below, the embedded frame buffer 702 can be configured to store pixel data in various RGB(A) formats.

As can be seen in Fig. 4, the processor interface 150 can be used, not only to supply data to the graphics pipeline 180, but also to enable the main processor (CPU) 110 to load data directly into the embedded frame buffer. This direct

10 loading of the embedded frame buffer by the CPU enables pixel formats other than RGB-type formats to be sent to the embedded frame buffer, thereby increasing the flexibility of the system to support a variety of applications. Specifically, the

processor interface 150 enables the main processor 110 to load pixel data in YUV format (i.e. luma/chroma format) into the embedded frame buffer from, for

15 example, an optical disk or other storage media. Once YUV format data is in the embedded frame buffer, it can be copied out to main memory in various texture formats, using the copy pipeline, for use as a texture by the texture environment unit (TEV) during a later rendering process. Thus, in accordance with the instant

invention, the embedded frame buffer is reconfigurable between various RGB(A) 20 formats and a YUV format. Each of these formats will be described in detail below.

Example RGB(A) Formats for the Embedded Frame Buffer

25 In this example, the embedded frame buffer (EFB) has a memory capacity of approximately 2MB. The maximum pixel width and height of the frame buffer is

determined by the size of each pixel. In accordance with the invention, and as shown in Figure 6, there are two different RGB pixel sizes that can be used for data in the embedded frame buffer 702. These sizes are:

- 5 • 48-bit color and Z; and
- 96-bit super-sampled color and Z

48-Bit Pixel Size Configuration

10 The 48-bit format for the embedded frame buffer (EFB) is preferably intended for non-anti-aliasing, and has the following features:

- 24-bit color (either 8/8/8 with no alpha, or 6/6/6/6 with 6 bits of alpha)
- 24-bit Z.

15 In this non-anti-aliasing mode, the 48-bit format can, in this example, support a maximum resolution of 640x528. Thus, the width must be between 0-640 and the EFB stride is fixed at 640 pixels. This non-antialiasing mode is based on a single point sample within each pixel. As indicated above, the point sample of each pixel is given a 24-bit color value and a 24-bit Z value. The color value
20 may have 8 bits each for red, green and blue values when no alpha is used, or it may have 6 bits each for red, green, blue and alpha.

96-Bit Pixel Size Configuration

25 The 96-bit super-sampling pixel format is preferably used for anti-aliasing and has the following features:

- 3 samples of 16-bit color (5 bits of Red, 6 bits of Green, 5 bits of Blue, no alpha)
- 3 samples of 16-bit Z (depth).

This 96-bit format can support a maximum resolution of 640x264. The
 5 width is preferably between 0-640 and the stride is fixed at 640.

In this example, the 96-bit format is used for anti-aliasing. However, this
 format may be used to achieve functionality other than anti-aliasing where three
 supersamples per pixel are desired. In other words, the 96-bit format provides the
 ability to store three samples (super-samples) for each pixel, as opposed to the
 10 single point sample per pixel as used in the 48-bit configuration.

As can be seen from the above, there are inherent tradeoffs between the 48-
 bit and 96-bit pixel formats. While the 96-bit anti-aliasing format enables an
 increase visual quality on the polygon edges and intersections, it does cost
 performance and Z quality. The 96 bit super-sampling EFB format requires twice
 15 as much memory as 48-bit point sampled pixels. This mode also reduces Z
 buffering precision to 16 bits rather than 24 bits in other formats. In this example,
 anti-aliasing also reduces peak fill rate from 800Mpixels/s to 400Mpixels/s.
 However, if more than one stage is employed in the texture environment unit
 (TEV), this reduction is hidden, in that, in this example, using two TEV stages
 20 also reduces the fill rate to 400Mpixels/s.

In one embodiment, the rendering rate with anti-aliasing activated drops
 down to two pixels/clock due to the embedded frame buffer 702 bandwidth
 limitations. However, if two or more textures are turned on, the rate at which
 pixel quads are sent to the pixel engine 700 drops down to less than or equal to
 25 one pixel quad every two clocks in this particular embodiment. In this case,
 turning on anti-aliasing will not impact fill rate. Thus, if a particular scene is

geometry-limited, then anti-aliasing will not adversely impact rendering performance. On the other hand, if a particular scene is fill-limited, rendering performance may be substantially adversely impacted by activating anti-aliasing as opposed to using the point sampled mode. The same application can activate and deactivate anti-aliasing for different scenes and different images depending on whether the scenes or images are geometry-limited or fill-limited – or depending upon the image quality required in a particular scene or image. The ability to dynamically change the pixel format in the frame buffer to, for example, activate and deactivate anti-aliasing, on a frame-by-frame basis provides great flexibility in allowing an application programmer to make tradeoffs between image quality and speed performance.

Anti-Aliasing Using the 96-Bit Configuration

15 The particular and preferred anti-aliasing methods and arrangements for use in connection with the instant invention are disclosed in commonly owned and co-pending application serial no. _____, filed _____ and entitled "Method and Apparatus For Anti-aliasing In A Grapics System", which is incorporated by reference herein in its entirety. A brief explanation of this anti-aliasing is provided below, in order to give a more complete understanding of the 96-bit pixel format for the embedded frame buffer.

In accordance with the anti-aliasing embodiment, anti-aliasing is performed in two main phases. In the first phase, data is loaded into the embedded frame buffer using three supersamples per pixel and the 96-bit pixel EFB format. In the second phase the data is read (or copied out) from the embedded frame buffer and further processed by the copy pipeline. The 96-bit format is particularly

advantageous for achieving the first phase of anti-aliasing, however, it can be used for any other suitable purpose as well.

In this example anti-aliasing use of the 96-bit format, the first anti-aliasing phase occurs when the rasterizer is performing edge rasterization into the

5 embedded frame buffer (EFB) 702. Preferably, this rasterizer is an edge and z rasterizer which generates x, y, z and coverage mask values for programmable super-sample locations within every visible pixel quad contained by the current triangle or other primitive being rendered. The pixel quads have three programmable subpixel locations within each pixel. The pixel quad includes 4
10 pixels in a 2x2 configuration. Within each pixel in the quad, three super-sample locations are programmably selected and specified. Preferably, the programmer can set the subsample locations by writing global registers. Since the location of each of the super-samples in each pixel is programmable in the example embodiment, the particular sampling locations for each quad can be changed as
15 desired for the particular application. On the other hand, in alternative embodiments, a particularly optimal multisample location values could be fixed (e.g., set in hardware) so the application programmer does not need to worry about it. Thus, while the locations are programmable in the example embodiment, a hardwired optional pattern could be used in lieu of programmability. Whatever
20 pattern is selected, it can be repeated across a certain number of neighboring pixels in a frame.

Once all of the primitives have been rendered for a frame, the embedded frame buffer will contain super-sampled pixel information for an entire frame. Due to the fact that three supersamples are provided for each pixel, and each
25 sample includes 16 bit color values and a 16 bit z value, each pixel is represented by 96-bits in the embedded frame buffer. The embedded frame buffer is then

ready for use by the copy pipeline, wherein the second phase of anti-aliasing can be performed. This second phase will be described in more detail below in connection with the copy pipeline. Further details regarding anti-aliasing are provided in the co-pending applications identified above.

5

YUV Embedded Frame Buffer Configuration

Figure 7 shows a further configuration for the embedded frame buffer 702 which is designed to store pixel data in YUV (luma/chroma) format which, for example, enables motion compensation under the MPEG standards (e.g. MPEG2) to be supported by the system. In this YUV configuration, the color buffer is preferably partitioned to store Y (720x576), U (360x288) and V (360x288) image planes for a YUV 4:2:0 frame. The partitioning of the color buffer preferably allocates as follows:

10

- 1024x640 8 bit Y image;
- 528x320 8 bit U image; and
- 528x320 8 bit V image.

15

20

The preferred location of the images are shown in Figure 7. The YUV data is preferably loaded into the embedded frame buffer by the main processor from an externally supplied medium, such as an optical disk or the like, or from any other suitable source other than the graphics pipe. It is noted, however, that the graphics pipeline can provide single component Y or U or V images, but it cannot draw three component YUV pixels. As will be explained in detail below, the YUV data in the embedded frame buffer can be further processed by the copy pipeline to

either display the data or to convert the data to texture data for subsequent use by the graphics pipeline.

Example Pixel Format Command

As explained above, the embedded frame buffer 702 can be selectively configured to support two RGB(A) pixel formats (48-bit and 96-bit) and a YUV format. The desired pixel format can preferably be set on a frame-by-frame basis using the API. An example API function for this purpose is as follows:

GXSetPixelFormat:

Argument:

GXPixelFormat	Format	//Sets pixel format for frame buffer
GXZCmprFormats	ZCmpr	//Sets compression format for 16 bit z
GXBool	Ztop	//Z compare before texture

This function sets the format of the embedded frame buffer. The function is called before any drawing operations are performed. The pixel format cannot be changed in the middle of a frame in the example embodiment. The 16 bit Z values (in multisample or anti-aliasing mode) can be uncompressed or compressed. The compressed values give a better precision and range. The Ztop flag can be used to perform depth comparisons before texture mapping (Z-before-texture). This can improve the texture bandwidth because less texels need to be fetched and filtered.

Interface Between the Pixel Engine and The Embedded Frame Buffer

An exemplary interface between the pixel engine 700 and the embedded frame buffer 702 is shown in Figure 8. Preferably, as shown in Figure 8, there are

4 copies of the embedded frame buffer (702a, 702b, 702c and 702d) -- 2 for color and 2 for Z. In this example, a read or write access to the embedded frame buffer from the pixel engine transfers 96 bits of data or 4 quads of color and Z. There are 4 address/control and read buses to the core of each of the buffers. The Z

- 5 channels A and B preferably share a write port 703a, and the color channels A and B preferably share a separate write port 703b. The embedded frame buffer preferably has enough bandwidth to blend 4 pixels per clock for peak fillrate of 800M pixels per second. The maximum size of the embedded frame buffer is 640 x 528 x 24b color and 24b Z. The embedded frame buffer is single-buffered and
- 10 expected to transfer a finished image to the external frame buffer for display.

Double buffered display is achieved in this manner. The address/control, read and write buses shown in Figure 8 are defined in the following table:

PE - EFB Interface

<u>Name:</u>	<u>Description:</u>
za_addr (16:0)	Z channel A quad address. There are 3 subfields: 3:0 column(3:0) valid range is 0 to 9 10:4 row(7:0) valid range 0 to 127 16:11 bank(5:0) valid range is 0 to 32
za_reb	Z change A read enable (active low).
za_web	Z channel A write enable (active low).
za_din(95:0)	Z channel A quad read bus. 4x24 bit Z for a quad. (23:0) Z for the upper left pixel in the quad (47:24) Z for the upper right pixel in the quad (63:48) Z for the lower left pixel in the quad (95:64) Z for the lower right pixel in the quad
zdout (95:0)	Z channels A and B quad Z write bus. 4x24 bit Z for the quad Refer to za_din for pixel locations on the bus
zb_addr (16:0)	Z channel B quad address (refer to za_addr for bit-fields)
zb_reb	Z channel B read enable (active low)

zb_web	Z channel B write enable (active low)
zb_din (95:0)	Z channel B quad read bus (refer to za_din for pixel locations)
ca_addr (16:0)	C channel A quad address. There are 3 subfields: 3:0 column(3:0) valid range is 0 to 9 10:4 row(7:0) valid range 0 to 127 16:11 bank(5:0) valid range is 0 to 32
ca_reb	Color channel A read enable (active low)
ca_web	Color channel A write enable (active low)
ca_din (95:0)	Color channel A quad read bus. 4x24 bit color for a quad. (23:0) color for the upper left pixel in the quad (47:24) color for the upper right pixel in the quad (63:48) color for the lower left pixel in the quad (95:64) color for the lower right pixel in the quad
Cdout (95:0)	Color channels A and B quad color write bus. 4x24 bit color for the quad Refer to ca_din for pixel locations on the bus.
cb_addr (16:0)	Color channel B quad address (refer to ca_addr for bit-fields)
cb_reb	Color channel B read enable (active low)
cb_web	Color channel B write enable (active low)
cb_din (95:0)	Color channel B quad read bus (refer to ca_din for pixel locations)

Example Copy Out Operations and Pipeline

- 5 Copy out operations, implemented in this example through what is referred to as the copy pipeline, is used to further process the pixel data from the embedded frame buffer (EFB) and to ultimately write the frame data in a selected format into the external frame buffer (XFB) 113 of main memory 112 as display data for display by the video interface or as texture data for later use by the graphics
- 10 pipeline (see Figure 11). RGB(A) or YUV420 data in the EFB can be copied out

to main memory YUV422, fields or frames. YUV422 data is copied out in scan-line order. There is a stride to allow skipping memory bytes between scan lines. Y8 is the lowest address, followed by U8, Y8 and V8. Copying in YUV format reduces the amount of memory used in main memory by 1/3.

5 A general block diagram of the copy pipeline, as it mainly relates to the processing of data from the EFB when in either of the two RGB(A) pixel configurations, is shown in Figure 9. As shown in Figure 9, this aspect of the copy pipeline includes an anti-alias/deflicker section 622, a gamma correction section 623, an RGB to YUV conversion section 624, and a Y scale section 626.

10 A more complete block diagram of the copy out pipeline for all EFB configurations (i.e. RGB(A) and YUV) is shown in Figure 12. Figure 11 shows the various paths that data can take between the various elements of the system. As shown in Figure 12, in order to reduce the amount of buffering needed for filtering operation in this example, a copy rectangle is broken into 32x32 tiles.

15 The tiles are double buffered by using two tile buffers 625a and 625b, so that while one tile is being filled, the other is being read and processed to be sent to main memory. A rectangle is preferably decomposed into tiles in the Y direction, followed by tiles in the X direction. Within a tile 4 pixels in a span are processed in one cycle. The pixel quads are processed in Y order followed by X.

20 Referring now more particularly to Figure 12, the particular copy pipeline operation depends on the particular configuration of the data in the embedded frame buffer and on the programmers desired result of the copy out operation. As explained above, the color buffer may contain RGB8, RGBA6, R5G6B5 (anti-aliasing) or YUV420 data, and the Z buffer may contain 24-bit Z or 3x16-bit Z

25 (anti-aliasing). Figure 12e shows a preferred conversion matrix for the various data formats available in this example.

If the embedded frame buffer is configured for and contains data in any of the RGB(A) formats describe above , the copy pipeline performs anti-

aliasing/deflickering operations, preferably using a programmable 7-tap vertical filter, as illustrated by block 628 in Figure 12. As described in greater detail

below, this filter blends the point sampled or supersampled (anti-aliasing) pixel data from multiple pixels and outputs a resulting pixel color. Once blended,

gamma correction is performed, as illustrated by block 623. An optional conversion can then be performed, if desired for the particular application, to

convert the RGB data to YUV444 format (block 641). If the data was converted

to YUV 444 at block 641, then another optional conversion can be performed at block 644 to convert from YUV444 to YUV422. If the copy pipe is being used for

display, this conversion to YUV422 can be performed to put the data in main memory display format, which is YUV422 in this example. The tile data is then

buffered at tile buffers 625a and 625b as explained above. Then, if the tile is

intended for display, Y scaling is performed at block 626, and the scaled data is copied out to main memory in YUV422 format (block 642) for use by the video

interface. On the other hand, if the data is intended to be used as a texture, Y scaling is not performed. Instead, the tile data is formatted into the desired texture

at block 640 and sent to main memory as a texture tile for possible use in a

subsequent graphics pipeline operation. The possible texture formats in this example are shown in Figure 12e and are listed below in connection with the

texture copy commands and register bit definitions.

If the embedded frame buffer is configured for and holds data in YUV420 format as described above, the copy pipeline has a slightly different operation as

shown by the lower portion of Figure 12. Specifically, the YUV420 data from the color buffer is first converted to YUV444 format (block 646). An optional

conversion from YUV444 to RGB can then be performed at block 648. The data path is then the same as described above with respect to the RGB(A) configurations. That is, the tile can be optionally converted, at block 644, to YUV422 (if previously converted to YUV444), then buffered and either scaled and sent to main memory as display data or converted to a desired texture format and stored as a texture tile in main memory. Figure 12f shows an example of how texture tiles (e.g. tiles 1-4) are stored in main memory.

Textures can be created by copying the Embedded Frame Buffer (EFB) to main memory using the GXCopyTex function. This is useful when creating dynamic shadow maps, environment maps, motion blur effects, etc.

All non-color index texture types except compressed textures (GX_TF_CMPR) can be created during the copy. The texture copy operation will create the correct tiling and formatting of the texture so it can be read directly by the hardware. Optionally, a box filter can be applied to the image in the EFB in order to create a lower level of detail (LOD) texture. The box filter can be used to create mipmaps from the EFB data. The following table shows exemplary texture copy formats and conversion notes.

Format	Conversion
GX_TF_I4	RGB -> (Y)UV, AA and non-AA pixel formats
GX_TF_I8	RGB -> (Y)UV, AA and non-AA pixel formats
GX_TF_A8	A (6 bits) -> A (8-bits, 2 MSBs replicated in LSBs), only with pixel format GX_PF_RGBA6_Z24
GX_TF_IA4	RGBA -> (Y)UV(A), if pixel format is not GX_PF_RGBA6_Z24, then A = 0xf
GX_TF_IA8	RGBA -> (Y)UV(A), if pixel format is not GX_PF_RGBA6_Z24, then A = 0xff

GX_TF_RGB565	RGB -> RGB, bits truncated for non-AA pixel formats.
GX_TF_RGB5A3	RGBA -> RGBA, if pixel format is not GX_PF_RGBA6_Z24, then MSB = 1. i.e., R5G5B5
GX_TF_RGBA8	RGBA -> RGBA, if pixel format is not GX_PF_RGBA6_Z24, then A = 0xff
GX_TF_Z24X8	Z (24 bits) -> Z (32 bits), only when pixel format is non-antialiased, GB_PF_RGB8_Z24 or GX_PF_RGBA6_Z24

Normally, the source and destination rectangles would have the same size. However, when copying small textures that will be composited into a larger texture the source and destination rectangles may differ. The format GX_TF_A8 is used specifically to copy the alpha channel from the EFB into a GX_TF_I8 formatted texture. The GX_TF_I8 will copy the luminance of the EFB into a GX_TF_I8 texture. When reading a texture, GX_TF_A8 and GX_TF_I8 are equivalent. When color textures are converted from an GX_PF_RGB8_Z24 pixel format to a lower-resolution color format, like GX_TF_RGB565, the least significant bits (LSBs) of the 8-bit colors are truncated. When color textures are converted from a lower resolution pixel format, such as GX_PF_RGB565_Z16, to a higher resolution texture format, such as GX_TF_RGB8, the most significant bits (MSBs) of each pixel are replicated in the LSBs of each texel. This conversion process distributes the estimation error evenly and allows each texel to represent the minimum or maximum value. In general, one should only copy textures containing alpha from an EFB with format GX_PF_RGBA6_Z24. When copying texture containing alpha from an EFB without alpha, alpha will be set to its maximum value. The GX_TF_Z24X8 format can be used to copy the 24-bit Z buffer to a 32-bit texture (equivalent format to GX_TF_RGBA8). To copy a

texture, the application preferably first allocates a buffer in main memory the size of the texture to be copied. This size can be determined using, for example, a GXGetTexBufferSize function. This function preferably takes into account texture padding and texture type in its calculations.

5 As can be seen from the above description, the copy out process in accordance with the instant invention enables various data formats to be used and various conversions/operations to be performed such that significant flexibility and functionality is provided thereby. By supporting YUV formats and enabling copy out as a texture, the copy pipeline line can be used to, for example, assist the
10 main processor in performing motion compensation. The copy out process as described above can be used not only to efficiently move and process data from the embedded frame buffer to the external frame buffer for display or as texture, but it also enables, for example, streaming video to be superimposed on a polygon by using the texture copy feature based on MPEG data which uses the YUV color
15 space.

Each of the copy pipeline operations described above are explained in greater detail below.

Exemplary Anti-Aliasing During Copy Out

20 As briefly explained above, when anti-aliasing is desired and the embedded frame buffer is configured for the 96-bit anti-aliased pixel data (e.g. R5G6B5 and Z16), a second stage of anti-aliasing can be performed during copy out. Specifically, the second stage of anti-aliasing is performed by the anti-
25 aliasing/deflicker section 622 during copy-out from the embedded frame buffer (EFB) 702 to the external frame buffer (XFB) 113.

The anti-aliasing/deflickering section 622 of the copy pipeline preferably applies a 7 tap vertical filter 628 (see Figure 12a) having programmable weightings (W0-W6) for each super-sample. The support for the vertical filter is preferably a three-vertical-pixel area. Thus, when determining color for a current pixel N in anti-aliasing mode, super-samples in the pixel immediately above the current pixel (N-1), and super-samples in the pixel immediately below the current pixel (N+1), as well as super-samples in the current pixel are preferably used. Preferably, the farthest sample from the current pixel within each of the two surrounding pixels is not used in the filtering operation. Thus, while the three pixel support for the filter has nine samples, only seven of the nine samples are used in the blending operation in the example embodiment. The resulting vertical filter output provides a single screen pixel color value (N') for eventual copying into the external frame buffer and display on display device 56.

In order to avoid the use of full line buffers, the copy operation preferably uses anti-aliasing (AA) buffering, wherein the copy is performed in strips of 32 pixels wide (X axis). The data-path for the strip-buffers in this exemplary AA buffering is shown in the block diagram of Figure 12b.

It is noted that additional details regarding anti-aliasing/de-flickering techniques and arrangements are provided in the commonly owned and co-pending application identified above. Inasmuch as this invention is directed to the embedded frame buffer and the overall copy out operation, regardless of the specific anti-aliasing operation used, further details regarding anti-aliasing are not provided herein.

Exemplary De-Flicker Filtering During Copy Out

The same vertical filter can be used during copy-out in a non-anti-aliasing mode to achieve a de-flickering function using point sampled pixels. In this mode, the sample patterns are not programmable. As shown in Figure 12c, the hardware uses only the center of the pixel as the sample locations. The weighting

5 coefficients (W0-W6) for each point sample are programmable as with the anti-aliasing filter. Thus, the vertical filter 628a in de-flickering mode uses three inputs (center only) from the current pixel and two inputs (center only) from each of the two vertically neighboring pixels, thereby obtaining the seven values for the filtering operation. The programmable weighting coefficients are applied to the

10 seven samples, and then the results are added to obtain the final pixel color (N'). Preferably, the de-flickering filter and AA filter are shared. The four strip buffers used in the AA data path (see Figure 12b) are also used to store quad strips. An exemplary block diagram of the data-path for de-flicker buffering is shown in Figure 12d. Further details regarding de-flickering are provided in the co-pending

15 application identified above.

Example RGB to YUV Conversion During Copy Out

A luma/chroma (YUV) format stores the same visual quality pixel as RGB,

20 but requires only two-thirds of the memory. Therefore, during the copy operation, the RGB(A) format in the EFB is converted to a YUV format in the XFB, in order to reduce the amount of main memory used for the external frame buffer (XFB). This conversion is done by the RGB to YUV section 624. An illustration of the conversion operation is shown in Figure 10a, wherein the RGB data is initially

25 converted to YUV444 format and then down-sampled to YUV 422 format for storage in the XFB as display data.

Vertical (Y) Scaling During Copy Out

The Y scale section 626 in the copy pipeline enables arbitrary scaling of a rendered image in the vertical direction. Horizontal scaling is preferably done during video display. A Y scale factor is defined in the API and determines the number of lines that will be copied, and can be used to compute the proper XFB size. A block diagram for the preferred vertical scaling in accordance with the instant invention is shown in Figure 10b. Vertical scaling is performed by using 8-bit lerps between 2 adjacent vertically adjacent strips. The lerp coefficient starts at 1.0. After a scan-line is outputted a fixed point (1.8) value is added to the lerp coefficient. The carry out of the lerp coefficients signals that a new scan-line is to be used. Two strip buffers 626a and 626b are used to keep 2 partial scan-lines that are on top of each other. Buffer A (626a) holds all incoming strips with even y value, while buffer B (626b) holds all the odd y value strips.

Gamma Correction During Copy Out

The gamma correction section 623 is used to correct for the non-linear response of the eye (and sometimes the monitor) to linear changes in color intensity values. Three choices of gamma may be provided (such as 1.0, 1.7 and 2.2). The default gamma is preferably 1.0 and is set in, for example, a GXInit command in the API.

Example Conversion Operations Usable During Copy Out

RGB to YCrCb(4:4:4) conversion

This conversion is used to reduce the amount of external frame-buffer needed by 1/3. The following equations are used for the conversion:

$$\begin{aligned} Y &= 0.257R + 0.504G + 0.098B + 16 \\ Cb &= -0.148R - 0.291G + 0.439B + 128 \\ Cr &= 0.439R - 0.368G - 0.071B + 128 \end{aligned}$$

Four pixels in a scan-line are converted in one cycle.

YCrCb(4:4:4) to YCrCb(4:2:2)

An illustration of the YUV444 to YUV422 conversion is shown in Figure

13. The following equations are used for this conversion:

$$\begin{aligned} c'(0,0) &= 1/4*c(0,0) + 1/2*c(0,0) + 1/4*c(1,0) \\ c'(2,0) &= 1/4*c(1,0) + 1/2*c(2,0) + 1/4*c(3,0) \\ c'(4,0) &= 1/4*c(3,0) + 1/2*c(4,0) + 1/4*c(5,0) \\ c'(m,n) &= 1/4*c(m-1,n) + 1/2*c(m,n) + 1/4*c(m+1,n) \end{aligned}$$

YCrCb(4:2:0) to YCrCb(4:4:4)

This conversion is done in two parts, as illustrated in Figures 14a and 14b.

The first part, shown in Figure 14a, converts from YUV420 format to YUV422 format. This conversion uses the following equations:

$$\begin{aligned} c'(0,0) &= 3/4*c(0,0.5) + 1/4*c(0,0.5) \\ c'(2,0) &= 3/4*c(2,0.5) + 1/4*c(2,0.5) \\ c'(4,0) &= 3/4*c(4,0.5) + 1/4*c(4,0.5) \\ c'(0,1) &= 3/4*c(0,0.5) + 1/4*c(0,2.5) \\ c'(2,1) &= 3/4*c(2,0.5) + 1/4*c(2,2.5) \\ c'(4,1) &= 3/4*c(4,0.5) + 1/4*c(4,2.5) \end{aligned}$$

$$\begin{aligned}
c'(0,2) &= 3/4 * c(0,2.5) + 1/4 * c(0,0.5) \\
c'(2,2) &= 3/4 * c(2,2.5) + 1/4 * c(2,0.5) \\
c'(4,2) &= 3/4 * c(4,2.5) + 1/4 * c(4,0.5) \\
c'(m,n) &= 3/4 * c(m,n-0.5) + 1/4 * c(m,n+1.5) \text{ for } n=\text{odd} \\
c'(m,n) &= 3/4 * c(m,n+0.5) + 1/4 * c(m,n-1.5) \text{ for } n=\text{even}
\end{aligned}$$

The second part of this YUV420 to YUV444 conversion, as shown in Figure 14b, up-samples the YUV422 data from part one above to YUV444. This conversion uses the following equations:

$$\begin{aligned}
c'(0,0) &= c(0,0) \\
c'(1,0) &= 1/2 * c(0,0) + 1/2 * c(2,0) \\
c'(2,0) &= c(2,0) \\
c'(3,0) &= 1/2 * c(2,0) + 1/2 * c(4,0) \\
c'(m,n) &= c(m,n) && m \text{ is odd} \\
c(m,n) &= 1/2 * c(m-1,n) + 1/2 * c(m+1,n) && m \text{ is even}
\end{aligned}$$

YcbCr (4:4:4) to RGB conversion

MPEG2 operates in YCbCr (YUV) color space. This conversion can be used to convert the YUV data to RGB data during copy out. The following equations are used for this conversions:

$$\begin{aligned}
R &= 1.164(Y-16) + 1.596 (Cr-128) \\
G &= 1.164(Y-16) - 0.813 (Cr-128) - 0.391(Cb-128) \\
B &= 1.164(Y-16) + 2.018(Cb-128)
\end{aligned}$$

Four pixels in a scan-line are converted in one cycle.

Example Copy Out Commands

The EFB source and destination of the display copy operation is specified using an exemplary API function as follows:

GXCopyFBToDisplay

5 Argument:

```

    u16  SrcLeft      //Upper-Left coordinate of the source rectangle
    u16  SrcTop
    u16  SrcWidth     //Width, in pixels, of the source rectangle
    u16  SrcHeight    //Height, in pixels, of the source rectangle
10    Void*DstBase     //Address of destination buffer in memory
    u16  DstStride    //Stride, in multiple of 32B, of destination buffer
    GXBool Clear      //enable clearing color and Z frame buffers

```

15 This function copies the contents of the embedded frame buffer (EFB) to the display buffer 113 in main memory. By the term “copy out” we don’t mean simply a transfer of all the information; rather, we mean that the contents of the embedded frame buffer are read out, further processed (e.g., filtered, resampled, scaled, etc.) and that the resulting data is then sent elsewhere (e.g., to an external point sample type frame buffer). The origin of the rectangle is defined by

20 SrcLeft(X) and SrcTop(Y). The Clear flag enables clearing of the color and z buffer to the current clear color and z values. The clearing of the embedded frame buffer preferably occurs simultaneously with the copy operation.

The following exemplary function can be used to set controls for the copy operations:

25

GXSetCopyControl

Arguments:

```

GXFbClamps    ClampFlags;    //Clamping flags for framebuffer filtering.
GXTexFormats  TexFormat;    //Format of texture (i.e. destination) pixels.

```

```

GXFbInterlace   Interlaced;    //Display buffer is interlaced (YUV422).
GXGamma         Gamma;        //Gamma correction on display buffer pixels.
u16               VertScale;    //vertical 1/scale value (1.8 format).
GXBool            MipFilter;    // Apply mipmap filter (texture copy only).

```

5

This function sets the controls used during copy from the embedded frame buffer to the display buffer. These controls are set once and then multiple copy commands can be issued.

The following is an exemplary API command for copying data out of the embedded frame buffer as a texture:

10

GXCopyFBToTexture

Argument:

```

u16      SrcLeft      //Upper-Left coordinates of the source rectangle.
15  u16      SrcTop;
u16      SrcWidth;   //Width, in pixels, of the source.
u16      SrcHeight; //Height, in pixels, of the source rectangle.
void*    DstBase;    //Address of destination buffer in memory (32B aligned).
u16      DstStride; //Stride, in multiple of 32 B, of destination buffer.
20  GXBool Clear;    //Enable clearing color and z framebuffers.

```

This function copies the embedded frame buffer to a texture image buffer in main memory in order to generate texture images for subsequent use by the graphics pipeline.

25

The following lists exemplary parameters that can be designated using the exemplary API commands above:

GXFbClamps:

```
GX_CLAMP_TOP,           //Clamp top edge of image for filtering.
GX_CLAMP_BOTTOM,        //Clamp bottom edge of image for filtering.
```

5 GXFbInterlace:

```
GX_INTLC_OFF           //Interlace is off.
GX_INTLC_EVEN          //Interlace even lines.
GX_INTLC_ODD           //Interlace odd lines.
```

GXZCmprFormats:

```

GX_ZC_LINEAR,          //Linear 16 bit z. No compression.
GX_ZC_14E2             //14e2 floating point format.
15 GX_ZC_13E3           //13e3 floating point format.

```

GXGamma:

```

20      GX_GM_1_0      //Gamma 1.0
      GX_GM_1_7      //Gamma 1.7
      GX_GM_2_2      //Gamma 2.2

```

GXTexFormats:

25	GX_TF_I4	//Intensity 4 bits
	GX_TF_I8	//Intensity 8 bits
	GX_TF_IA4	//Intensity-Alpha 8 bit (44)
	GX_TF_IA8	//Intensity-Alpha 16 bit (88)
	GX_TF_C4	//Color Index 4 bit
30	GX_TF_C8	//Color Index 8 bit
	GX_TF_CA4	//Color Index + Alpha 8 bit (44)
	GX_TF_C6A2	//Color Index + Alpha 8 bit (62)
	GX_TF_CA8	//Color Index + Alpha 16 bit (88)
	GX_TF_R5G6B5	//RGB 16 bit (565)
35	GX_TF_RGB5A1	//RGB 16 bit (5551)
	GX_TF_RGBA8	//RGB 32 bit (8888)
	GX_TF_CMPR	//Compressed 4 bits/texel. RGB8A1.

GXPixelFormats:

```

GX_PF_RGB8_Z24
GX_PF_RGBA6_Z24
GX_PF_RGB565_Z16
5  GX_PF_Z24           //used for z buffer copy (diagnostics only)
GX_PF_Y8
GX_PF_U8
GX_PF_V8
GX_PF_YUV420         //used for YUV copy.

```

10

Example Pixel Engine Registers

Figures 15-17 show exemplary registers used by the pixel engine in connection with the copy out operations. Specifically, Figure 15 shows an exemplary control register. The bit definitions for this exemplary control register are as follows:

15

Control Register Bits:

20	2:0 pixtype	5:3 zcmode	000: linear z compression for 16 bit Z
	000: RGB8/Z24		001: 14e2 z compression for 16 bit Z
	001:RGBA6/Z24		010: 13e3 z compression for 16 bit Z
	010:RGB_AA/Z16		011: 12e4 z compression for 16 bit Z
	011:Z (for copying Z buffer as texture		100: inverted_linear z compression for 16 bit Z
25	100: Y8 or U8 or V8		101: inverted_14e2 z compression for 16 bit Z
	101: YUV 420 (only used for copy operation)		110: inverted_13e3 z compression for 16 bit Z
			111: inverted_12e4 z compression for 16 bit Z
		6: ztop	0: z at the end of the pipe
			1: z buffering before texture mapping

30

Bits 0-2 designate the pixel type for the copy operation. Writing to this control register causes the graphics pipe stages between the edge rasterizer (RAS0) and the pixel engine (PE) to be flushed. In this example, this will can take a minimum of 90 cycles. Writing to this register can also be used to sync up copy texture with rendering a primitive that uses the texture.

35

Figure 16 shows an exemplary register for the copy to texture operation. The bit definitions for this exemplary register are as follows:

Texture Copy Register Bits:

5	1:0 src_clamp							
	x1:clamp top							
	1x:clamp bottom							
10	2 color conversion							
	0:no color conversion							
	1:convert RGB to YUV							
	6:3 tex_format							
	pixtype: rgb8,rgba6,rgb_aa	yuv8	yuv8	yuv8	yuv420	yuv420	z	
	yuvsel: x	x	Y	U	V	x	x	x
15	ccv_mode : OFF	ON	OFF	OFF	OFF	OFF	ON	OFF
	0000:R4	Y4	Y4	U4	V4	Y4	R4	z[23:20]
	0001:R8	Y8	Y8	U8	V8	Y8	R8	z[23:16]
	0010:RA4	YA4	YA4	UA4	VA4	YA4	RA4	
	0011:RA8	YA8	YA8	UA8	VA8	YA8	RA8	
20	0100:R5G6B5	Y5U6V5	Y5Y6Y5	U5U6U5	V5V6V5	Y5Y6Y5	R5R6B5	
	0101:RGB5A3	YUV5A3	YYY5A3	UUU5A3	VVV5A3	YUV5A3	RGB5A3	
	0110:RGBA8	YUVA8	YYA8	UUUA8	VVVA8	YUVA8	RGVA8	z[23:00],0xff
	0111:A8	A8	0xff	0xff	0xff	0xff	0xff	0xff
	1000:R8	Y8	Y8	U8	V8	Y8	R8	z[23:16]
25	1001:G8	U8	Y8	U8	V8	U8	G8	z[15:08]
	1010:B8	V8	Y8	U8	V8	V8	B8	z[07:00]
	1011:RG8	YU8	YY8	UU8	VV8	YU8	RG8	z[23:16]
	(red as intensity and Green as alpha)							
30	1100 GB8	UV8	YY8	UU8	VV8	UV8	GB8	z[15:00]

8:7 gamma (only when arc_format is any of the RGB formats)

00: gamma = 1.0

01: gamma = 1.7

10: gamma = 2.2

11: reserved

9: mip_map_filter

0: no filtering (1:1)

1: box filtering (2:1)

11: clr (should be set to 0 for rgb_aa)

- 0: do not clear Z and Color efb
- 1: clear Z and color efb

13:12 intlc

- 00: progressive
- 01: reserved
- 10: interlaced (even lines)
- 11: interlaced (odd lines)

16:15 ccv_mode

- 0x: automatic color conversion, based on pixtype and texture format
- 10: color conversion off (rgb to yuv)
- 11: color conversion on (rgb to yuv)

The pixel types allowed for this operation are RGB8, RGBA6, RGB_AA (i.e. anti-aliasing (R5G6B5)), YUV8 and YUV420. Bits 3-6 determine the format of the texture stored by the copy command in texture buffer. In this example, this texture formatting is done in the texture format section 640 shown in Figure 12. In this example, Clr is not supported for pixel type YUV420 and should be set to 0. Gamma correction is also not supported for pixel type YUV420 in this example.

Figure 17 shows an exemplary register for the copy to display operation. The bit definitions for this exemplary register are as follows:

Display Copy Register Bits:

- | | |
|--|---|
| 1:0 src_clamp | 10:scen |
| x1: clamp top | 0: no vertical scaling |
| 1x: clamp bottom | 1: vertical scaling |
| 8:7 gamma (only when src_format is any of the RGB formats) | 11:clr (should be set to zero for rgb_aa) |
| | 0: do not clear Z and Color efb |
| | 1: clear Z and color efb |

00: gamma = 1.0

01 :gamma = 1.7

10: gamma = 2.2

11: reserved

13:12 intlc

00: progressive

01: reserved

10: interlaced (even lines)

11: interlaced (odd lines)

The pixel types allowed for this operation are RGB8, RGBA6, RGB_AA (anti-aliasing) and YUV420. Clr is not supported for pixel type YUV420 and should be set to 0. Gamma correction is also not supported for pixel type YUV420 in this example.

Other Example Compatible Implementations

Certain of the above-described system components 50 could be implemented as other than the home video game console configuration described above. For example, one could run graphics application or other software written for system 50 on a platform with a different configuration that emulates system 50 or is otherwise compatible with it. If the other platform can successfully emulate, simulate and/or provide some or all of the hardware and software resources of system 50, then the other platform will be able to successfully execute the software.

As one example, an emulator may provide a hardware and/or software configuration (platform) that is different from the hardware and/or software configuration (platform) of system 50. The emulator system might include software and/or hardware components that emulate or simulate some or all of hardware and/or software components of the system for which the application software was written. For example, the emulator system could comprise a general purpose digital computer such as a personal computer, which executes a software emulator program that simulates the hardware and/or firmware of system 50.

Some general purpose digital computers (e.g., IBM or MacIntosh personal computers and compatibles) are now equipped with 3D graphics cards that provide 3D graphics pipelines compliant with DirectX or other standard 3D graphics command APIs. They may also be equipped with stereophonic sound cards that provide high quality stereophonic sound based on a standard set of sound commands. Such multimedia-hardware-equipped personal computers running emulator software may have sufficient performance to approximate the graphics and sound performance of system 50. Emulator software controls the hardware resources on the personal computer platform to simulate the processing, 3D graphics, sound, peripheral and other capabilities of the home video game console platform for which the game programmer wrote the game software.

Figure 18A illustrates an example overall emulation process using a host platform 1201, an emulator component 1303, and a game software executable binary image provided on a storage medium 62. Host 1201 may be a general or special purpose digital computing device such as, for example, a personal computer, a video game console, or any other platform with sufficient computing power. Emulator 1303 may be software and/or hardware that runs on host platform 1201, and provides a real-time conversion of commands, data and other information from storage medium 62 into a form that can be processed by host 1201. For example, emulator 1303 fetches "source" binary-image program instructions intended for execution by system 50 from storage medium 62 and converts these program instructions to a target format that can be executed or otherwise processed by host 1201.

As one example, in the case where the software is written for execution on a platform using an IBM PowerPC or other specific processor and the host 1201 is a personal computer using a different (e.g., Intel) processor, emulator 1303 fetches

one or a sequence of binary-image program instructions from storage medium 62 and converts these program instructions to one or more equivalent Intel binary-image program instructions. The emulator 1303 also fetches and/or generates graphics commands and audio commands intended for processing by the graphics and audio processor 114, and converts these commands into a format or formats that can be processed by hardware and/or software graphics and audio processing resources available on host 1201. As one example, emulator 1303 may convert these commands into commands that can be processed by specific graphics and/or sound hardware of the host 1201 (e.g., using standard DirectX, OpenGL and/or sound APIs).

An emulator 1303 used to provide some or all of the features of the video game system described above may also be provided with a graphic user interface (GUI) that simplifies or automates the selection of various options and screen modes for games run using the emulator. In one example, such an emulator 1303 may further include enhanced functionality as compared with the host platform for which the software was originally intended.

Figure 18B illustrates an emulation host system 1201 suitable for use with emulator 1303. System 1201 includes a processing unit 1203 and a system memory 1205. A system bus 1207 couples various system components including system memory 1205 to processing unit 1203. System bus 1207 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. System memory 1207 includes read only memory (ROM) 1252 and random access memory (RAM) 1254. A basic input/output system (BIOS) 1256, containing the basic routines that help to transfer information between elements within personal computer system 1201, such as during start-up, is stored in the ROM 1252.

System 1201 further includes various drives and associated computer-readable media. A hard disk drive 1209 reads from and writes to a (typically fixed) magnetic hard disk 1211. An additional (possible optional) magnetic disk drive 1213 reads from and writes to a removable "floppy" or other magnetic disk 1215.

5 An optical disk drive 1217 reads from and, in some configurations, writes to a removable optical disk 1219 such as a CD ROM or other optical media. Hard disk drive 1209 and optical disk drive 1217 are connected to system bus 1207 by a hard disk drive interface 1221 and an optical drive interface 1225, respectively. The drives and their associated computer-readable media provide nonvolatile storage
10 of computer-readable instructions, data structures, program modules, game programs and other data for personal computer system 1201. In other configurations, other types of computer-readable media that can store data that is accessible by a computer (e.g., magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, random access memories (RAMs), read only
15 memories (ROMs) and the like) may also be used.

A number of program modules including emulator 1303 may be stored on the hard disk 1211, removable magnetic disk 1215, optical disk 1219 and/or the ROM 1252 and/or the RAM 1254 of system memory 1205. Such program modules may include an operating system providing graphics and sound APIs,
20 one or more application programs, other program modules, program data and game data. A user may enter commands and information into personal computer system 1201 through input devices such as a keyboard 1227, pointing device 1229, microphones, joysticks, game controllers, satellite dishes, scanners, or the like. These and other input devices can be connected to processing unit 1203 through a
25 serial port interface 1231 that is coupled to system bus 1207, but may be connected by other interfaces, such as a parallel port, game port Fire wire bus or a

universal serial bus (USB). A monitor 1233 or other type of display device is also connected to system bus 1207 via an interface, such as a video adapter 1235.

System 1201 may also include a modem 1154 or other network interface means for establishing communications over a network 1152 such as the Internet.

5 Modem 1154, which may be internal or external, is connected to system bus 123 via serial port interface 1231. A network interface 1156 may also be provided for allowing system 1201 to communicate with a remote computing device 1150 (e.g., another system 1201) via a local area network 1158 (or such communication may be via wide area network 1152 or other communications path such as dial-up or
10 other communications means). System 1201 will typically include other peripheral output devices, such as printers and other standard peripheral devices.

In one example, video adapter 1235 may include a 3D graphics pipeline chip set providing fast 3D graphics rendering in response to 3D graphics commands issued based on a standard 3D graphics application programmer
15 interface such as Microsoft's DirectX 7.0 or other version. A set of stereo loudspeakers 1237 is also connected to system bus 1207 via a sound generating interface such as a conventional "sound card" providing hardware and embedded software support for generating high quality stereophonic sound based on sound commands provided by bus 1207. These hardware capabilities allow system 1201
20 to provide sufficient graphics and sound speed performance to play software stored in storage medium 62.

While the invention has been described in connection with what is presently considered to be the most practical and preferred embodiment, it is to be understood that the invention is not to be limited to the disclosed embodiment, but

[illegible]